LA PROGRAMMAZIONE ORIENTATA AGLI OGGETTI IN PYTHON 3

POLITECNICO OPEN UNIX LABS, CORSI PYTHON 2012

Sommario. Questo documento copre gli argomenti trattati nella lezione 3 del corso, integrando le slide con annotazioni fatte a voce durante la lezione. E' sempre possibile rivedere le slide e il commento originale dal canale Youtube del POuL. Rispetto ai materiali mostrati durante il corso, queste sono state riviste, corrette e integrate e arricchite di collegamenti ipertestuali di vita più o meno corta che troverete evidenziati con un piacevole colore magenta.

- Riferimenti e link utili:
  - Pagina del corso con tutti i materiali
  - La documentazione ufficiale di Python 3
  - Le domande più comuni riguardo Pyhton su Stack Overflow
- Tool utili:
  - iPython (se usate Debian, Ubuntu o derivate: sudo apt-get install ipython3)

Indice

1. Cos'è un "oggetto"

1.1. **La definizione.** La definizione da libro di testo di un "tipo di dato astratto", preso dalle relative slide del prof. Luciano Baresi per il corso di Ingegneria del Software, cita:

Definizione. Un *tipo di dato astratto* (A.D.T.) è una astrazione sui dati che non li classifica in base alla loro rappresentazione, ma in base al loro comportamento atteso espresso in termini di un insieme di operazioni applicabili a quei dati.

In parole povere, un ADT è una combinazione di dati e operazioni sugli stessi.

Esempio. Un *numero complesso* ha:

- dati: parte reale, parte immaginaria
- operazioni: moltiplicazione, divisione, modulo, anomalia

Un *file* ha:

- dati: nome, posizione
- operazioni: scrittura, lettura, spostamento

Una *anatra* ha:

- dati: numero di riconoscimento, posizione
- operazioni: camminare, nuotare, starnazzare

1.2. **Classe e istanza.** Una *classe* descrive i dati dell'A.D.T. e come manipolarli. È lo stampo che viene usato per tutti gli oggetti dello stesso tipo.

Esempio. Esempi di classi sono:

- Persona; Impiegato; Studente; Professore; Dottorando; Erasmus; ...
- Animale; Anatra; Cartone animato; Protagonista; ...

Una *istanza* di una classe è, invece uno specifico valore che una classe può assumere.

Esempio. Esempi di istanze sono:

- Santi; Nicola; Alessandro; Radu; ...
- Duffy Duck; Paperino; Fufi; Lassie; ...

I *metodi* sono funzioni che operano sulle singole istanze. Vengono definiti all'interno della classe perchè sono uguali per tutte le istanze, ma operano sulle istanze[1].

1.3. **In pratica.** La sintassi di definizione di una classe e dei suoi metodi è come segue.

```
1  class Duck():
2    def __init__(self, name = "Duffy"):
3      self.name = name
4    def quack(self):
5     print("{.name} Duck says Quack!".format(self))
6
7  paperino = Duck("Donald")
8  paperino.quack()
```

[1]Non sempre: i metodi "statici" operano sulle classi stesse e sono preceduti da @classmethod. I metodi statici operano su dati statici, associati alla classe, non alle istanze, e il loro primo parametro è la classe stessa. Il loro uso in Python è però limitato.

Commenti riga per riga:

(1) I metodi sono semplicemente dichiarati come funzioni innestate dentro una dichiarazione class.
(2) Il primo parametro dei metodi è sempre l'istanza su cui viene chiamato il metodo. Questo parametro viene "sempre" chiamato self, ma è importante sottolineare il fatto che è la posizione, non il nome del parametro che conta. In particolare, questo parametro non viene "mai" passato in modo esplicito.
(3) I dati delle istanze non sono esplicitati nella classe stessa: basta assegnare nuovi attributi all'oggetto, come visto in riga 3.
(4) Questo metodo non prende parametri: chi chiama il metodo non passa self esplicitamente.
(5) Questa è una forma comoda di format per accedere direttamente agli attributi di un oggetto. Questa riga è equivalente a:
```
print(self.name + " Duck says Quack!")
```
(6) La guida stilistica di Python vorrebbe qui due righe e una riga prima della riga 2 e della riga 4. Per motivi di impaginazione, però, mi trovo costretto a "risparmiare" sul whitespace. Spero mi perdonerete :)
(7) Questa sintassi crea una istanza di Duck. I parametri passano al metodo particolare in riga 2, chiamato __init__ e detto in gergo "*costruttore*." Il costruttore di una classe inizializza i dati delle relative istanze; in questo caso assegna a paperino.name il valore "Donald".
(8) Questa sintassi chiama il metodo quack sull'oggetto paperino prima creato, ottenendo quindi "Donald Duck says Quack!". Notate come noi non passiamo esplicitamente il parametro self: è cura del linguaggio trasformare "paperino.quack()" in "Duck.quack(paperino)".

Esempio. Ad esempio, nelle lezioni precedenti abbiamo visto come costruire un'agenda usando semplici funzioni:

```
agenda = { "pippo": "0129133337",
           "pluto": "0369313337",
           "minni": "0926424242" }

def trova(agenda, contatto):
  return agenda[contatto]
```

Rendere questo codice "a oggetti" è semplice:

```
class Agenda
  def __init__(self, contatti: 'Un dict tra nomi e numeri'):
    self.persone = contatti
  def trova(self, contatto):
    return self.persone[contatto]

agenda = Agenda({ "pippo": "0129133337",
                  "pluto": "0369313337",
                  "minni": "0926424242" })
```

2. Duck Typing

In molti linguaggi prima si pensa a cosa si vuole rappresentare e poi lo si rappresenta in una combinazione di dati e operazioni. Le operazioni rimangono quindi legati a tali combinazioni di dati e operazioni, cioè classi specificati per nome. Creare una propria classe che sia compatibile con un'altra ma leggermente diversa e usarla al suo posto diventa quindi complicato.

In Python le cose sono diverse: sono i dati e le operazioni a determinare che cosa si sta rappresentando in Python. L'identità della classe che vi sta dietro non è quindi altrettanto importante. Considerate ad esempio la seguente funzione, da non confondere con la funzione predefinita sum():

```
1 def sum_of(*items):
2   result = items[0]
3   for item in items[1:]:
4     result = result + item
5   return item
```

Osservazione. La sintassi in riga 1 indica una funzione di un numero arbitrario di argomenti. Gli argomenti vengono passati all'interno della lista items. Ad esempio, chiamare sum_of() porta a items == []; chiamare sum_of(a, b, c) darà invece items == [a, b, c].[2] Notate che sum(), invece, vuole un solo parametro: una lista di numeri.

Quel che questa funzione fa è prendere il primo elemento (riga 2), aggiungendovi poi gli altri elementi ad uno ad uno (righe 3-4). Cos'è, esattamente che stiamo sommando, però? Qualsiasi cosa che può essere sommata, ecco cosa:

```
sum_of(2, 3, 5)        # -> 10
sum_of("2", "3", "5") # -> "235"
sum_of([2], [3, 5])    # -> [2, 3, 5]
sum_of(2, b)           # TypeError
```

Dopo tutto, se i nostri parametri possono essere sommati... ci interessa davvero sapere che cosa sono? La nostra funzione prova a sommarli e basta: se possibile, bene, altrimenti viene portata un'eccezione. A noi non interessa che oggetti ci troviamo a gestire se questi si comportano nel modo in cui ci aspettiamo si comportino (in questo caso ci aspettiamo che possano essere sommati).

Citando la frase di dubbia origine che dà nome a questa "filosofia" di programmazione:

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> *Se vedo un volatile e quel volatile starnazza come un'anatra e nuota come un'anatra e starnazza come un'anatra, io chiamo quel volatile un'anatra.*

Ci interessa veramente se in realtà era uno svasso?

Esprimere lo stesso codice in altri linguaggi come il Java[3] non è semplice. Servirebbe generare al volo una interface implicita con i metodi utilizzati dalla funzione. Qualcosa così:

---

[2] E' possibile fare anche l'inverso e passare gli argomenti a una funzione tramite una lista. In altre parole, sum_of(*range(3)) è equivalente a sum_of(0, 1, 2). L'uso di questa sintassi non si ferma alle chiamate di funzione; per esercizio provate a vedere come si comporta questo assegnamento: primo, *altri = range(3). Per maggiori perversioni sul tema, guardate queste due sezioni del tutorial sul controllo di flusso in Python 3.

[3] Se non conoscete il Java saltate tranquillamente alla prossima sezione.

```
public interface Summable {
  public Summable sum(Summable other);
}

class SumTools(){ //?
  static Summable sum_of(Collection<Summable> items) {
    ...
  }
}
```

Ovviamente però bisognerebbe aggiungere l'interfaccia Summable a tutti gli oggetti che supportano la somma, cosa impossibile: in Java (come in Python, del resto) le classi predefinite sono immutabili e non è possibile quindi usare sum_of() su Integer o anche solo int senza quantità generose di polimorfismo.

Si guadagna quindi in termini di *flessibilità*: sum_of in Python funziona con *tutti* i tipi che supportano la somma. Si perde però in termini di *controlli di correttezza*: se qualcosa non va, ce ne possiamo accorgere solo quand'è troppo tardi: quando il programma ti esplode in mano.

3. Metodi "speciali"

__init__ non è l'unico metodo "speciale" in Python. Tutti gli operatori (+, −, *, <, in, ...) sono dietro le quinte metodi con nomi standard predefiniti. Ad esempio:

```
"a".__add__("b")          # <=> "a" + "b"
(5).__mul__(4)            # <=> 5 * 4
"foo".__contains__("o") # <=> "o" in "foo"
print.__call__("!!")      # <=> print("!!")
```

Provate ad applicare la funzione predefinita dir() su oggetti come numeri, liste, dizionari: essa ne elenca gli attributi. Vedrete molte cose in comune. Ma non finisce qui: noi possiamo creare le nostre classi con i nostri metodi. Definendoli opportunamente possiamo permettere alle nostre classi di essere sommate, chiamate, iterate o hashate[4].

Consideriamo ad esempio un'implementazione di un vettore bidimensionale. Sarebbe triste dover usare un metodo .add() per fare operazioni basilari come sommare due vettori:

```
Vector2D(4, 4).add(Vector2D(2, −2)) #ugh!
```

...e infatti basta definire il proprio metodo __add__:

```
class Vector2D():
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __add__(self, other):
    return Vector2D(self.x + other.x, self.y + other.y)
```

Possiamo poi usare questa classe così:

[4]Un oggetto hashabile può essere usato come chiave in un dizionario

```
a = Vector2D(4, 4)
b = Vector2D(2, -2)
c = a + b # -> Vector2D(6, 2)
```

Questo vi dà un'enorme potere sul linguaggio stesso e può permettervi di creare elaborati linguaggi specifici di linguaggio, come Lepl[5]. Attenzione però a non abusare di questo potere!

> With great power there must also come great responsibility!
> *Con un grande potere vengono anche grandi responsabilità.*

Ad esempio, usare len() su un oggetto Vector2D per ottenerne la lunghezza *si può fare*:

```
class Vector2D():
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __len__(self):
    return (this.x**2 + this.y**2)**.5
```

...ma *sarebbe meglio* avere un metodo tradizionale, a.length(), per il semplice fatto che nessun altro oggetto in Python si comporta così! In Python len() dà il numero di oggetti in un contenitore; considerare un vettore un'insieme di vettorini di lunghezza unitaria (e restituire un numero non intero) è veramente arrampicarsi sugli specchi.

Se volete che Persona più Persona dia Matrimonio, che Matrimonio meno Persona dia un Divorzio che porti la Persona a perdere custodia di suo Figlio, può anche andar bene... ma usiamo parsimonia: è molto meglio lasciare gli operatori matematici ai concetti matematici, le funzioni dei contenitori ai contenitori e usare i metodi normali per i casi normali. Se implementare gli operatori, però, può migliorare la leggibilità di tutto il codice su cui state lavorando: ben venga!

3.1. **Esempio: Vector.** Ad esempio, consideriamo un'estensione n-dimensionale di Vector2D; chiamiamola Vector. Nel caso generico il costruttore dovrà essere così:

```
class Vector():
  def __init__(self, components):
    self.proj = components
  def __add__(self, other):
    components = []
    for xi, yi in zip(self.proj, other.proj):
      components.append(xi + yi)
    return Vector(components)
```

Osservazione. zip è una funzione predefinita che funziona così:

$$\mathrm{zip}\left([a_1,\ldots,a_n],[b_1,\ldots,b_n]\right) = [(a_1,b_1),\ldots,(a_n,b_n)]$$

Infatti noi, qui, vogliamo operare su ciascuna componente dei due vettori, sommandoli.

Non possiamo fare altrimenti senza sapere a priori quante dimensioni ha il nostro vettore! Usare una classe così sarebbe però davvero scomodo. Ad esempio, consideriamo di voler implementare il prodotto vettoriale tridimensionale. Wikipedia dice:

[5]Di nuovo, non spaventatevi se non comprendete questi termini! Sono argomenti molto avanzati.

Sia $\mathbf{a} = a_1\mathbf{i} + a_2\mathbf{j} + a_3\mathbf{k}$, $\mathbf{b} = b_1\mathbf{i} + b_2\mathbf{j} + b_3\mathbf{k}$. Allora:

$$\mathbf{a} \times \mathbf{b} = (a_2b_3 - a_3b_2)\,\mathbf{i} + (a_3b_1 - a_1b_3)\,\mathbf{j} + (a_1b_2 - a_2b_1)\,\mathbf{k}$$

Ricordandoci che la formula conta a partire da 1 ma Python, come ogni linguaggio di programmazione che si rispetti, inizia a contare da 0, dovremo aver cura di ricopiare la formula sopra riportata ricordandoci di decrementare ogni pedice... in bocca al lupo!

Oppure potremmo fare così:[6]

```
class Vector():
  #...
  def __getitem__(self, index):
    return self.proj[index - 1]
  def cross_mult(a, b):
    return Vector(a[2] * b[3] - a[3] * b[2],
                  a[3] * b[1] - a[1] * b[3],
                  a[1] * b[2] - a[2] * b[1])
```

E' molto più facile ora assicurarsi di aver implementato correttamente questo metodo. Con __getitem__ implementiamo la possibilità di accedere a un elemento di un vettore direttamente come se fosse una tupla, ma con indici a base 1. Cosa succede tuttavia a chi questo non lo sa e prova ad accedere all'elemento zero? Egli accederà all'indice -1, il che non genera errore: l'elemento -1 è l'ultimo (l'elemento -2 è il penultimo e così via). Potremmo risolvere il problema generando un'eccezione a mano qualora l'utente della classe usi un indice uguale o inferiore a zero.

```
class Vector():
  #...
  def __getitem__(self, index):
    if index < 1:
      raise IndexError("Indexes are 1-based. Consider using .proj directly
          instead.")
    return self.proj[index - 1]
```

In alternativa potremmo prendere una seconda strada diversa, "creando" attributi al volo:

```
class Vector():
  #...
  shorthands = {'x': 0, 'i': 0,
                'y': 1, 'j': 1,
                'z': 2, 'k': 2}
  def __getattr__(self, name):
    try:
      return self.proj[shorthands[name]]
    except KeyError as exc:
      raise AttributeError from exc
  def cross_mult(a, b):
    return Vector(a.y * b.z - a.z * b.y,
```

[6] Sì, lo so: cross_mult dovrebbe avere self come primo parametro, non a. Spero potrete perdonarmi :)

```
        a.z * b.x - a.x * b.z,
        a.x * b.y - a.y * b.x)
```

La funzione __getattr__ viene chiamata ogniqualvolta si tenta di accedere a un attributo che la classe altrimenti non ha. Il suo nome viene quindi passato come parametro a questa funzione, che si occupa di restituire il valore necessario o generare un'eccezione AttributeError.

Notate ora come, tra le altre cose, Vector possa essere usato tramite Duck Typing al posto di Vector2D. Il codice scritto con Vector2D in mente legge direttamente gli attributi x e y; qui li abbiamo ri-implementati senza effettivamente allocare loro alcuna memoria. In aggiunta è semplice ora accedere direttamente a ogni componente del vettore.

Il costo in velocità è relativo: sì, shorthands[7] è un dizionario e accedervi ha un costo, ma in realtà tutti gli accessi ad attributi avvengono tramite dizionari dietro le quinte.

Osservazione. Qui cambiamo con un costrutto try-except il tipo di eccezione da KeyError ad AttributeError perchè è AttributeError l'errore che gli oggetti in Python restituiscono quando si tenta di accedere a un attributo che non esiste. Il fatto che noi usiamo dietro le quinte un dizionario che genera eccezioni KeyError non riguarda chi la classe la usa. L'aggiunta della sintassi "as exc"/"from exc" in Python 3 aiuta a fare ciò senza perdere tutte le informazioni su dove l'eccezione era stata effettivamente originata.

Non abbiamo ancora finito però! Manca un pezzo fondamentale per un vettore: la relazione di uguaglianza.

```
class Vector():
  #...
  def __eq__(this, that):
    for tit, tat in zip(this.proj, that.proj):
      if tit != tat:
        return true
    return false
```

Python non deduce dalla relazione di uguaglianza quella di disuguaglianza, purtroppo. Poco male:

```
class Vector():
  #...
  def __ne__(this, that):
    return not Vector.__eq__(this, that)
```

Python in realtà avrebbe un modo più furbo di operare, tramite la funzione __cmp__(x, y). Questa funzione restituisce 1 se vogliamo che x > y, 0 se vogliamo che x == y e -1 se vogliamo che x < y; da questa funzione Python deduce anche se x != y. Tutte queste operazioni sono anche definibili singolarmente, ma __cmp__ permette di crearle tutte in un solo colpo e in modo coerente. Il problema, ovviamente, è che non ha matematicamente senso dire che un vettore è maggiore o minore di un altro.

[7] Notate, tra parentesi, che shorthands è quella che chiameremmo una variabile *statica*: è associata alla classe, non alla singola istanza.

Osservazione. Il codice riportato in questi esempi è semplificato per chiarezza espositiva. Ad esempio, tutte le varianti di cross_mult dovrebbero esplicitamente controllare che i vettori siano effettivamente tridimensionali, visto che questo prodotto non è altrimenti definito. La nostra __eq__ facilmente darà falsi negativi a causa di come i computer gesticono i numeri decimali.

3.2. **Iteratori fatti in casa.** Facciamo un'altro esempio per metter luce su un'altro modo in cui le nostre classi possono comportarsi come cittadini di prima classe in Python: l'iterazione. Consideriamo ad esempio di voler costruire una tavola di valori di $\sin x$ per $x = -\pi, -\pi + \frac{\pi}{10}, \ldots, +\pi$. Non possiamo usare la classe inbuilt range per generare questo insieme: range opera solo su interi. Creare una classe che operi anche sui numeri a virgola mobile, tuttavia, è facile:

```
1  class RangeFloat(object):
2    def __init__(self, start, stop, step = 1):
3      self.start = start
4      self.stop = stop
5      self.step = step
6    def __iter__(self):
7      result = self.start
8      while result < self.stop:
9        yield result
10       result += self.step
11
12 from math import sin, pi
13 for tick in RangeFloat(-pi, pi, pi/10):
14   print("sin({}) = {}".format(tick, sin(tick)))
```

La "magia" avviene alla riga 9, dove abbiamo usato la parola chiave yield. Una funzione che usa yield invece di return è detta *generatore*. Python può convertire automaticamente un generatore in un *iteratore*, un accrocchio di non comoda scrittura che però viene usato dai cicli for-in di Python come quello in riga 13. In pratica, la funzione __iter__ viene eseguita fino a raggiungere la yield (righe 7-9) questa restituisce l'elemento successivo su cui iterare e il controllo passa al ciclo (riga 14). Terminato il giro di ciclo il controllo ritorna a __iter__, la quale viene ripristinata al suo stato precedente e riprende l'esecuzione dalla operazione stessa di yield (cioè, righe 10, 8, 9).

A un certo punto la condizione del ciclo while (riga 8) sarà falsa e la funzione fa un return implicito. Questo segnala che il ciclo for-in può essere terminato per esaurimento degli oggetti su cui iterare.

Osservazione. Che ci crediate o no, abbiamo guardato qua solo ad alcuni dei "metodi speciali": la sezione 3.4 della documentazione di Python ha un elenco più completo, incluse funzioni di conversione quali __str__, __repr__, __bool__, ...

Appendice A. Python Funzionale

Python eredita da linguaggi LISP alcuni costrutti per gestire in modo più semplice e descrittivo liste di dati. Talvolta questo modo di procedere è detto "programmazione funzionale", perchè basato sul concetto di "funzioni di ordine superiore" o funzioni di funzioni. E' anche la base del sistema di "map reduce" usato con successo da aziende come Google. Non lasciatevi spaventare però dai nomi altisonanti, però: non è niente di eccezionale.

I programmi spendono circa il ~90% del tempo di CPU in iterazioni su liste di dati, applicando le stesse operazioni – o, se volete, la stessa *funzione* – su ciascuno di essi. Il modo imperativo di fare questo è attraverso cicli: ad esempio, cicli for-in o while. Questo stesso concetto si esprime i ntermini di programmazione funzionale con i concetti di map e reduce, gli stessi concetti su cui Google ha costruito la sua rete di calcolo distribuito.

Consideriamo un esempio pratico: il calcolo della media armonica di n (arbitario) dati:

$$H(x_1, x_2, \ldots, x_n) = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}$$

Con un paradigma imperativo probabilmente scriveremmo:

```
def harmonic_avg(numbers):
  result = 0
  for i in numbers:
    result += 1/i
  return len(numbers) / result
```

Non c'è niente di *sbagliato* nel scrivere così la funzione, intendiamoci. E' possibile però scrivere qualcosa che più si avvicina alle nostre intenzioni: il numero di elementi diviso la *somma* di ciascun elemento *invertito*. Noi prendiamo ogni elemento, applichiamo a ciascuno la stessa operazione di reciproco e poi ne otteniamo un singolo valore: la somma di tutti gli elementi. Per esplicitare questo processo possiamo riscrivere così la funzione:

```
1 def harmonic_avg(numbers):
2   result = []
3   for i in numbers:
4     result.append(1/i)
5   return len(result) / sum(result)
```

Abbiamo separato le due fasi: nelle righe 2-4 applichiamo l'operazione di reciproco sulla lista numbers, ottenendo una lista ottenuta dalla prima tramite la trasformazione $f(x) = \frac{1}{x}$. Notiamo però che abbiamo aumentato, e non di poco, la quantità di spazio e memoria occupata dalla nostra funzione, visto che creiamo in memoria la lista result.

Ad ogni modo, questa operazione nel paradigma funzionale è detta di *map*: esiste infatti una funzione map che applica una funzione a ogni elemento di una lista:

$$\text{map}\left(f, (x_1, x_2 \ldots)\right) = \left(f(x_1), f(x_2), \ldots\right)$$

Se nel nostro caso $f(x) = \frac{1}{x}$, avremo $\left(\frac{1}{x_1}, \frac{1}{x_2}, \ldots\right)$. In codice questo diventa:

```
def harmonic_avg(numbers):
  def one_over(number):
    return 1/number
  result = map(one_over, numbers)
  return len(numbers) / sum(result)
```

Dietro le quinte, map usa dei generatori: gli elementi della lista vengono solo valutati quando richiesti. Questo significa che non incorriamo più nei costi aggiuntivi (e inutili) per creare una lista

intermedia result. Se però quello che ci interessa è una lista, possiamo ottenerla semplicemente usando il costruttore della lista sul risultato: list(result).[8]

Questa scrittura rimane però un pò troppo verbosa per i nostri gusti. Esiste un modo più veloce per definire funzioni semplici che eseguono semplicemente una return, ed è offerta dalla parola chiave lambda. Questa è la sintassi:

```
lambda x: 1/x
```

Questa è una funzione anonima (cioè, appunto, senza nome) di un parametro x che ritorna il valore 1/x. Questa funzione può essere assegnata a una variabile:

```
def harmonic_avg(numbers):
    one_over = lambda x: 1/x
    result = map(one_over, numbers)
    return len(numbers) / sum(result)
```

...o passata direttamente per argomento:

```
def harmonic_avg(numbers):
    result = map(lambda x: 1/x, numbers)
    return len(numbers) / sum(result)
```

Se pensate che questa scrittura sia ancora troppo verbosa o macchinosa, Python ha ancora un'altro asso nella manica: la *comprehension*. Le comprehensions permettono di costriuire liste, insiemi o dizionari a partire da una lista e una operazione di mapping. Vediamo solo la comprehension di liste[9] (o *list comprehension*), tenendo conto che non c'è niente di concettualmente diverso quando si parla di dizionari o set. Questa la sintassi:

```
def harmonic_avg(numbers):
    result = (1/x for x in numbers)
    return len(numbers) / sum(result)
```

La list comprehension è a destra del segno di uguale in riga 2. Questa prende la lista numbers e, ad ogni elemento x, vi sostituisce 1/x.

Osservazione. Tutte le scritture supportano una sintassi estesa che, in aggiunta all'operazione di map, produce anche quella di *filter*, in cui applichiamo la funzione solo ad alcuni degli elementi nella lista iniziale. Nel nostro esempio, la funzione darà un errore se numbers contiene uno o più zeri. Se li vogliamo ignorare basta estendere la list comprehension così:

```
result = (1/x for x in numbers if x != 0)
```

Così facendo, se numbers è [3, 5, 0], otterremo come valore di result questo: [1/3, 1/5].

La stessa list comprehension può essere passata come parametro:

---

[8] A lezione ho usato erroneamente len(result) invece di len(numbers). Non si può usare len() direttamente su un generatore.

[9] Per i più curiosi, ecco tutte le comprehensions supportate da Python 3. Provatele:

```
list_comprehension = (1/x for x in numbers) # => iterator
list_comprehension = [1/x for x in numbers) # => list
set_comprehension = {1/x for x in numbers} # => set
dict_comprehension = {x: 1/x for x in numbers} # => dict
```

```
def harmonic_avg(numbers):
    return len(numbers) / sum(1/x for x in numbers)
```

ma in questo caso è forse meglio *non* farlo per motivi di leggibilità. Gli spaghetti li vogliamo solo nei piatti, non nel codice.

Quindi ecco la nostra media armonica in tutto il suo splendore funzionale:

```
def harmonic_avg(numbers):
    result = (1/x for x in numbers)
    return len(numbers) / sum(result)
```

Notiamo come sia banale tradurre in termini matematici questa funzione:

$$H(\mathbf{x}) = \frac{\|\mathbf{x}\|}{\sum_{i \in \mathbf{x}} \frac{1}{i}}$$

L'espressione matematica legge "La media armonica di un vettore $\mathbf{x}$ è la sua dimensione fratto la sommatoria di 1 su i per ogni i elemento di $\mathbf{x}$." Il nostro codice legge "la media armonica di una lista `numbers` è la sua dimensione fratto la sommatoria di `1/x` per ogni `x` in `numbers`." E' facile notare che queste due espressioni facciano la stessa cosa, ma non solo: la esprimono negli stessi termini!

Quindi, rispetto alla funzione imperativa della stessa funzione abbiamo scritto:

- meno righe di codice
- codice che meglio esprime quel che vogliamo ottenere
- codice in questo caso leggermente più lento.

Come al solito, però, è bene esercitare giudizio nell'uso di queste feature, senza abusarne. Ad esempio, potremmo riscrivere la funzione sopra citata come:

```
harmonic_avg = lambda x: len(x) / sum(1/y for y in x)
```

ma questo avrebbe senso *solo* per funzioni effettivamente usa e getta (o nel code golfing[10]).

[10] Il divertente esercizio quasi da enigmistica che consiste nello scrivere un programma che faccia X usando meno caratteri possibili.